Partitioning Program into Hardware and Software

Shengchao Qin*
Department of Informatics
School of Mathematical Sciences
Peking University, Beijing, 100871
qinshc@pubms.pku.edu.cn

Jifeng He†
International Institute for Software Technology
The United Nations University
P.O.Box 3058, Macau
jifeng@iist.unu.edu

Abstract

Hardware and software co-design is a design technique which delivers computer systems comprising hardware and software components. A critical phase of co-design process is to decompose a program into hardware and software. This paper proposes an algebraic partitioning method whose correctness is verified in the algebra of programs. We introduce the program analysis phase before program partitioning and develop a collection of syntax-based splitting rules, where the former provides the information for moving operations from software to hardware and reducing the interaction between components, and the latter supports a compositional approach to the program partitioning.

1. Introduction

The design of a complex software product like a nuclear reactor control system is ideally decomposed into a progression of related phases. It starts with an investigation of the properties and behaviours of the process evolving within its environment, and an analysis of requirement for its safety performance. From these is derived a specification of the electronic or program-centered components of the system. The project then may go through a series of design phases, ending in a program expressed in a high level language. After translation into a machine code of the chosen computer, it is executed at high speed by electronic circuitry. In order to achieve the time performance required by the customer, additional application-specific hardware devices may be needed to embed the computer into the system which it controls.

With chip size reaching one million transistors, the complexity of VLSI algorithms is approaching that of software algorithms. However, the design methods for circuits resemble the low level machine language programming methods. Selecting individual gates and registers in a circuit like selecting individual machine instruction in a program. State transition diagrams are like flowcharts. These methods may have been adequate for small circuit design when they were introduced, but they are not adequate for circuits that perform complicated algorithms. Industry interest in the formal verification of embedded systems is gaining ground since an error in a widely used hardware device can have significant repercussions on the stock value of the company concerned. In principle, proof of correctness of a digital device can always be achieved by making a comparison of the behavioral description of the circuit with its specification. But for a large system this would be impossibly laborious. What we need is a useful collection of proven equations and other theorems, which can be used to calculate, manipulate and transform the specification formulae to the product.

Hardware/software co-design is a design technique which delivers computer systems comprising hardware and software components. A critical phase of co-design process is to partition a program into hardware and software. This paper proposes a partitioning method whose correctness is verified using the algebraic laws developed for the high level programming language. To meet performance goals, and reduce the communication between components, our approach combines the program analysis technique with the syntax-based splitting rules to move heavy-weight operations from software to hardware. The allocation of variables is also based on the data flow analysis of the source program. One of the advantages of our method is the integration of the splitting phase with the joining phase of the partitioning process. It optimizes the underlying target architecture, and facilitates the reuse of hardware devices.

The algebraic approach advocated in this paper to verify the correctness of the partitioning process has been successfully employed in the **ProCoS** project on "Provably Correct Systems". The original **ProCoS** project [6] concentrated almost exclusively on the verification of standard compiler

*Partially supported by NNSFC No. 69873003
†On leave from East China Normal University

Proceedings of the Eighth Asia-Pacific Software Engineering Conference (APSEC'01)
1530-1362/01 $17.00 © 2001 **IEEE**

of a high-level programming language based on Occam down to a microprocessor based on Transputer [5]. Sampaio showed how to reduce the compiler design task to one of program transformation; his formal framework is also a procedural language and its algebraic laws [14]. Towards the end of the first phase of the project, Ian Page *et al* made rapid advance in the development of hardware compilation technique using an Occam-like language targeted towards Field Programmable Gate Arrays [11], and He Jifeng *et al* provided a formal verification of the hardware compilation scheme within the algebra of Occam programs [4].

Recently, some works have suggested the use of formal methods for the partitioning process [1, 2, 15]. Balboni *et al* adopt Occam as an internal model for the system exploration and partitioning strategy. Cheung pursues the structural transformation and verification within the functional programming framework. However, neither has provided a formal proof for the correctness of the partitioning process. In [15], Silva *et al* provide a formal strategy for carrying out the splitting phase automatically, and presents an algebraic proof for its correctness. However, the splitting phase delivers a large number of simple processes, and leaves the hard task of clustering these processes into hardware and software components to the clustering phase and the joining phase. Furthermore, additional channels and local variables introduced in the splitting phase to accommodate huge number of parallel processes actually increase the data flow between the hardware and software components.

The remainder of this paper is organized as follows. Section 2 describes the splitting strategy. Section 3 introduces the programming language we adopt and explores its algebraic laws. Section 4 poses the static analysis that we perform on the source program. Section 5 investigates the underlying target architecture of hardware/software components. Section 6 provides the syntax-based hardware/software splitting rules in both bottom-up and top-down styles.

2. Splitting Strategy

This section describes our partitioning strategy. A sequential source program of a communication language is generated from the customer's requirements. A static analysis [10] is performed on the source program in order to provide to the programmer statistical data, such as structural complexities of expressions and their occurrence frequencies, distributive information with respect to those variables occurring in expressions. Based on the result of the analysis, the programmer marks those parts of the program that are worth to be implemented by hardware and leaves others to software, and as well divides the interface of the program to two disjoint parts.

The implementation-oriented program marking and interface (variable) partitioning are conducted by the following guidelines:

- For the concern of security or other special reasons, some specific blocks will be predetermined to be implemented by hardware or software.
- In general, those procedures which are frequently invoked and those specific blocks that occurs frequently should be marked out to be implemented by hardware, to gain high performances.
- Some procedures/blocks involving very complicated computation (e.g., containing intricate expressions) should be marked and implemented by hardware, to improve timing performance.
- Busy variables should be allocated to hardware, to make high-speed access available, whereas the remaining variables and large scale data structures, such as large arrays, should be left to software, to achieve lower costs.
- The number of interactions between software and hardware should be minimized since they incur high costs.
- In addition, the customer's demands concerned with the performance and the cost should also be taken into account.

We take such a marked source program as input of our hardware/software splitting algorithm that generates as output a program comprising two concurrent processes representing software and hardware components respectively.

3. Preliminaries

The language we select to perform hardware/software partitioning is a subset of Occam which was designed for constructing communicating systems.

1. Sequential Process:

$$
\begin{array}{rcl}
S & ::= & PC \text{ (primitive command)} \\
& | & S; S \text{ (sequential composition)} \\
& | & \mathit{if}\ b\ S\ \mathit{else}\ S \text{ (conditional)} \\
& | & S \sqcap S \text{ (non-deterministic choice)} \\
& | & b * S \text{(iteration)} \\
& | & (g\ S) [\!] (g\ S) \text{ (guarded choice)} \\
& | & \mathit{declaration} \bullet S \text{ (local declaration)}
\end{array}
$$

where $PC ::= (x := e) \mid \mathit{skip} \mid \mathit{chaos} \mid c!e \mid c?x$
$\mid \mathit{procn}(e, v)$ (procedure invocation)
$\mid \langle S \rangle$ (annotated block)

and g is skip or a communication event $c!e$ or $d?x$.

2. Parallel Program:

$$P ::= S \mid P \parallel P \mid \mathit{declaration} \bullet P$$

where $\mathit{declaration} ::= \mathit{var\_dec} \mid \mathit{chan\_dec} \mid \mathit{proc\_dec}$
$\mathit{var\_dec} ::= \mathit{var}\ v : \mathbf{type}(v)$

chan_dec $::=$ *chan* $c : \mathbf{type}(c)$
proc_dec $::=$
procedure procn$(\mathbf{in}\ u : \mathbf{type}(u),\ \mathbf{out}\ v : \mathbf{type}(v))$
$\mathbf{begin}\ S\ \mathbf{end}$ □

In the later discussion, we adopt *Var*(P) and *Chan*(P) to denote the set of variables and channels employed by P. Moreover, we will not mention the type information of a variable in a declaration if it is obvious.

As a subset of Occam, the language enjoys a rich set of algebraic laws presented in [13, 3, 7, 9, 8]. Here we only explore those algebraic laws which will be employed within the proofs in the following sections.

Successive assignments to the same variable can be combined to one assignment.

L1 $x := e; x := f \ = \ x := f[e/x]$

Sequential composition is associative, and has left zero $chaos$ and unit *skip*. It distributes backward over internal and external choices and conditional.

L2 $(P; Q); R \ = \ P; (Q; R)$
L3 $chaos; P \ = \ chaos$
L4 $skip; P \ = \ P; skip \ = \ P$
L5 $(P \sqcap Q); R \ = \ (P; R) \sqcap (Q; R)$
L6 $(g\ P) [\!] (h\ Q); R \ = \ (g\ (P; R)) [\!] (h\ (Q; R))$
L7 $(if\ b\ P\ else\ Q); R \ = \ if\ b\ (P; R)\ else\ (Q; R)$

Assignment distributes forward over conditional.

L8 $v := e; (if\ b(v)\ P\ else\ Q) \ =$
$if\ b(e)\ (v := e; P)\ else\ (v := e; Q)$

The input and output event can be renamed as follows.

L9 $c ? x \ = \ \mathbf{var}\ lx \bullet (c ? lx; x := lx)$
L10 $c ! e \ = \ \mathbf{var} lx \bullet (lx := e; c ! lx)$

Iteration is subject to the fixed point theorem.

L11 $b * P \ = \ if\ b\ (P; b * P)\ else\ skip$

Parallel operator is symmetric and associative, and has $chaos$ as zero.

L12 $P \parallel Q \ = \ Q \parallel P$
L13 $P \parallel (Q \parallel R) \ = \ (P \parallel Q) \parallel R$
L14 $chaos \parallel P \ = \ chaos$

Parallel operator also distributes over conditional. It's disjunctive.

L15 $(if\ b\ P\ else\ Q) \parallel R \ = \ if\ b\ (P \parallel R)\ else\ (Q \parallel R)$, provided *Var*$(b) \cap$ *Var*$(R) \ = \ \emptyset$.
L16 $(P \sqcap Q) \parallel R \ = \ (P \parallel R) \sqcap (Q \parallel R)$

Local variable declaration enjoys the following laws.

L17 $\mathbf{var}\ x \bullet (x := e) \ = \ skip$, provided x does not occur in e.
L18 $\mathbf{var}\ x \bullet (if\ b\ P\ else\ Q) \ =$
$if\ b\ (\mathbf{var}\ x \bullet P)\ else\ (\mathbf{var} x \bullet Q)$,
provided x is not free in b.
L19 $\mathbf{var}\ x \bullet (P; Q) \ = \ P; \mathbf{var} x \bullet Q$, provided x is not free in P.
L20 $\mathbf{var} x \bullet (P; Q) \ = \ \mathbf{var} x \bullet P;\ Q$, provided x is not free in Q.

The following law deals with assignment expansion.

L21 $(x := e; S) \parallel T \ = \ x := e; (S \parallel T)$

The following law is one of the general expansion laws of Occam [13], which deals with the case where two parallel processes are guarded choice constructs.

L22 Let $P \ = \ [\!]_{i=1}^{n}\ (g_i\ P_i)$, $Q \ = \ [\!]_{j=1}^{m}\ (h_j\ Q_j)$, where each $g_i,\ h_j$ has one of the forms $c ! e$, $c ? x$ or *skip*, then $P \parallel Q \ = \ [\!]_{r=1}^{N}\ (k_r\ R_r)$, where the pairs $< k_r, R_r >$ are precisely all possibilities from the following:
(i) $R_r = P_i \parallel Q$ and ($k_r = g_i = skip$ or $k_r = g_i = c ! e$ or $k_r = g_i = c ? x$), where $c \notin$ *Chan*(Q);
(ii) $R_r = P \parallel Q_j$ and ($k_r = h_j = skip$ or $k_r = h_j = c ! e$ or $k_r = h_j = c ? x$), where $c \notin$ *Chan*(P);
(iii) $R_r = x := e; (P_i \parallel Q_j)$ and $k_r = skip$ and (($g_i = c ! e$ and $h_j = c ? x$) or ($g_i = c ? x$ and $h_j = c ! e$)).

Corollary 3.1
(C1) $(c ! e; P) \parallel (c ? x; Q) \ = \ x := e; (P \parallel Q)$
(C2) Let $P \ = \ (c ! e;\ P_1)$, $Q \ = \ (Q_1;\ Q_2)$, where $c \in$ *Chan*(Q), but no channel in *Chan*$(P) \cap$ *Chan*(Q) occurs in Q_1, then $P \parallel Q \ = \ Q_1;\ (P \parallel Q_2)$.
(C3) Let $P \ = \ (S_1; c ? x; S_2)$, and $Q \ = \ (T_1; c ! e; T_2)$, where neither S_1 nor T_1 mentions channels in *Chan*$(P) \cap$ *Chan*(Q), then
$P \parallel Q = (S_1 \parallel T_1); ((c ? x; S_2) \parallel (c ! e; T_2))$.

The proof is presented in [12].

We exhibit two derived algebraic laws as follows from those basic ones.

The test of conditional should be evaluated first.

DL1 $if\ b\ P\ else\ Q \ = \ \mathbf{var} lb \bullet (lb := b\ ;\ if\ lb\ P\ else\ Q)$
Proof RHS $\{(L8)\}$
$= \mathbf{var} lb \bullet (if\ b\ (lb := b; P)\ else\ (lb := b; Q))$ $\{(L18)\}$
$= if\ b\ (\mathbf{var} lb \bullet (lb := b; P))\ else\ (\mathbf{var}\ lb \bullet (lb := b; Q))$ $\{(L20)\}$
$= if\ b\ (\mathbf{var} lb \bullet (lb := b); P)\ else\ (\mathbf{var}\ lb \bullet (lb := b); Q)$ $\{(L17)\}$
$= LHS$ □

The condition of iteration is evaluated at the beginning of every loop.

DL2 $b * P \ = \ \mathbf{var} lb \bullet (lb := b\ ;\ lb * (P\ ;\ lb := b))$

The proof is omitted here because of the page limit. It can be found in [12].

We introduce an ordering relation between two programs as follows before further discussion.

Definition 3.3 (Refinement)
Given programs $P,\ Q$, we say Q is a refinement of P, denoted as $P \sqsubseteq Q$, if $P \sqcap Q = P$ is algebraically provable.

4. The Static Analysis

This section illustrates the static analysis on the source program, which provides plenty of information to the programmer to assist the appropriate implementation-oriented program marking and interface partitioning of the source

program, aiming to gain higher performance and as well achieve lower cost.

The static analysis comprises two parts: the subprogram/expression analysis and the variable analysis. The output of the subprogram/expression analysis consists of three kinds of information, which will be presented in three tables, respectively.

- Structural complexity of non-trivial expressions in the program and numbers of their occurrences
- Numbers of invocations of procedures, the complexity of their parameters and their structures
- Complexity of those implementation-undetermined blocks

The complexity of expressions is specified by the function $complex$ as follows.

Definition 4.1 Let *EXP* be the set of expressions occurred in the source program, $complex : EXP \rightarrow N$ is inductively defined on the structure of expressions:

$complex(v) =_{df} w(\mathbf{type}(v))$, for any variable v;
$complex(c) =_{df} 1$, for any constant c; and
$complex(\mathbf{op}(e_1, \ldots, e_n)) =_{df}$
$\Sigma_{i=1}^{n} complex(e_i) + complex(\mathbf{op})$,

where $\mathbf{op}$ is any operator used to construct expressions in the source language, and $complex(\mathbf{op})$ is defined by the programmer in accordance to the complexity of $\mathbf{op}$, the function $w : TYPE \rightarrow N$ associates a number to each type of variables and channels in the program to measure their complexity. □

By scanning the program, we obtain the occurrence frequency of expressions, which can be regarded as another factor of criteria about busyness of expressions.

By scanning the program, we also gain the number of invocations of procedures. Through analysing the declarations of those procedures in the program, we get the complexity of their parameters. Suppose $v_1 : T_1, \ldots, v_k : T_k$ is the list of parameters for some procedure, then the complexity of its parameters is $\Sigma_{i=1}^{k} w(T_i)$.

It is also possible to define the complexity of procedures or blocks that do not contain iterations. If the number of loops can be predicted or estimated, the complexity of those which contain iterations can also be calculated.

Definition 4.2 The complexity of subprograms (procedures/blocks) can be evaluated as follows.

$com(v := e) =_{df} w(\mathbf{type}(v)) + w(:=) + complex(e)$;
$com(c\,!\,e) =_{df} w(\mathbf{type}(c)) + complex(e)$;
$com(c\,?\,x) =_{df} w(\mathbf{type}(c)) + complex(x)$;
$com(S_1; S_2) =_{df} com(S_1) + com(S_2)$;
$com(\mathit{if}\ b\ S_1\ \mathit{else}\ S_2) =_{df}$
$com(b) + \mathbf{max}(com(S_1), com(S_2))$;
$com((g_1; S_1) \,[\!]\, (g_2; S_2)) =_{df}$
$\mathbf{max}(com(g_1; S_1), com(g_2; S_2))$;
$com(\mathit{while}\ b\ S) =_{df}$
$\begin{cases} (com(b) + com(S)) \times \mathit{maxN},\ \mathit{num.\ of\ loops\ is\ maxN}; \\ \infty,\ \text{otherwise}. \end{cases}$

where $w(:=)$ is defined by the programmer. □

Based on the three tables the analysis generates, the programmer can appropriately figure out those parts that should be implemented by hardware, in accordance with those guidelines listed before.

The second step of the analysis provides the following information about variables.

- the structural complexity of each variable
- the occurrence frequency of each variable
- the distributive information of each variable

We illustrate an industrial example in the following.

Example 4.3 The source program is concerned with the design of an ATM Switch. The code is illustrated in the appendix.

Provided $complex(+) = complex(-) = complex(\leq) = complex(<) = 10$, $complex(\wedge) = complex(\neg) = 5$, $w(\mathbf{int}) = 8$, $w(\mathbf{Bool}) = 1$, $w(:=) = 2$, then the results of the analysis are listed below.

expression	*complex*	*num*
$RP + 1$	19	2
$RC + 1$	19	5
$AC + 1$	19	4
$RM + 1$	19	3
$1.\neg ok.LPP$	6	1
$2.ok.HPP \wedge ok.HPM$	7	1
$3.ok.HPP \wedge \neg ok.HPM \wedge ok.LPM$	18	1
$4.ok.HPP \wedge \neg ok.HPM \wedge \neg ok.LPM$	23	1
$5.\neg ok.HPP \wedge ok.LPP \wedge ok.LPM$	18	1
$6.\neg ok.HPP \wedge \neg ok.LPP$	17	1
$7.\neg ok.HPP \wedge \neg ok.LPP \wedge \neg ok.LPM$	28	1

procedure	*num*	*complex. of para.*	*complex. of proc.*
GCRA	4	57	141
UPT	9	82	94

variable	*num.*	*complex.*	*distribution*
VPI,VCI	2	8	input
GFC,PT,HEC,Pl,QoS	4	8	input/output
aT	5	8	input
X,L,LCT	5	32	GCRA, input
I	4	32	GCRA, input
nX	8	32	GCRA, output
nLCT	7	32	GCRA, output
ok	22	4	GCRA, 1,2,3,4,5,6,7
CLP	6	4	UPT, input
nCLP	12	4	UPT, input/output
RP,RM,RC,AC	10	8	UPT, input
nRP,nRM,nRC,nAC	9	8	UPT
send	10	1	UPT
nVPI,nVCI,pN	6	24	input/output

The criterion of the interface partitioning is that a variable should be allocated to hardware if its structure is not complicated and it occurs in those procedures/blocks which are assigned to hardware more often than those ones that are left to software.

5. The Hardware/software Target Architecture

This section describes the target architecture of our partitioning approach which confines hardware and software components to specially chosen forms. To synchronize their activities, we introduce a simple handshaking protocol to streamline communications between them.

Suppose $B = \{r_j, a_j \mid j \in I\}$ is a set of channels, we define $CP(B)$ as the set of communicating processes C with *Chan*$(C) \supseteq B$ and one of the following forms.

(1). a communicating process which does not use any channel in B.

(2). $r_j\,!\,e; C; a_j\,?\,x$, where C is a member of $CP(B)$ not interacting via channels in B.

(3). $C_1; C_2$, or $C_1 \sqcap C_2$, or $\mathit{if}\ b\ C_1\ \mathit{else}\ C_2$, or $(g_1\ C_1)[\!](g_2\ C_2)$, where both g_i and C_i lie in $CP(B)$, for $i = 1, 2$.

(4). $b * C$, where C is a member of $CP(B)$.

To simplify the interface design, we confine the interactions between the hardware and software components to the communications along the channels from the set B. Our partitioning rules will select the software components from the set $CP(B)$, and organise the hardware component in the form of

$D = \mu X \bullet ([\!]_{j\in I}(r_j?x_j; M_j; a_j!y_j; X)[\!]\mathit{skip})$

where none of M_j mentions channels in B. The communicating process D represents a digital device which offers a set of services to its environment, each of which responds to a request from its environment on an input channel r_j by running the corresponding program M_j and delivering the result to the output channel a_j afterwards. The translation from such a hardware specification to netlists will be tackled using the hardware compilation techniques [11].

We denote as $H(B)$ the set of those processes which own the same form as D.

Theorem 5.1 $(C_1; C_2) \parallel D = (C_1 \parallel D); (C_2 \parallel D)$, for any C_1, C_2 in *CP(B)*.

Proof By structural induction on C_1.

(1). No channels in *B* appear in C_1.

LHS $\{Co3.1(C2)\}$
$= C_1\ ;\ (C_2 \parallel D)$ $\{Co3.1(C2)\}$
$= RHS$

(2). $C_1 = r_j\,!\,e; C; a_j\,?\,x$, for some $r_j, a_j \in B, C \in$ *CP(B)*, and no channel in *B* occurs in *C*.

LHS $\{Co3.1(C1)\}$
$= x_j := e; ((C; a_j\,?\,x; C_2) \parallel (M_j; a_j\,!\,y_j; D))$ $\{Co3.1(C3)\}$
$= x_j := e; (C \parallel M_j); ((a_j\,?\,x; C_2) \parallel (a_j\,!\,y_j; D))$ $\{Co3.1(C1)\}$
$= x_j := e; (C \parallel M_j); x := y_j; (C_2 \parallel D)$ $\{Co3.1(C1, C3)\}$
$= RHS$

(3).$C_1 = C_{01}; C_{02}$, where $C_{01}, C_{02} \in$ *CP(B)*.

We know $C_{02}; C_2 \in$ *CP(B)*, from the definition of *CP(B)*. Then

LHS $\{hypothesis\}$
$= (C_{01} \parallel D); ((C_{02}; C_2) \parallel D)$ $\{hypothesis\}$
$= (C_{01} \parallel D); (C_{02} \parallel D); (C_2 \parallel D)$ $\{hypothesis\}$
$= RHS$

(4). C_1 is one of the cases: (i) $\mathit{if}\ b\ C_{01}\ \mathit{else}\ C_{02}$, (ii) $C_{01} \sqcap C_{02}$, (iii) $(g_1\ C_{01})[\!](g_2\ C_{02})$, where $C_{01}, C_{02} \in CP(B)$. We demonstrate the first case here, others are similar ([12]).

LHS $\{(L7)\}$
$= (\mathit{if}\ b\ (C_{01}; C_2)\ \mathit{else}\ (C_{02}; C_2)) \parallel D$ $\{(L15)\}$
$= \mathit{if}\ b\ ((C_{01}; C_2) \parallel D)\ \mathit{else}\ ((C_{02}; C_2) \parallel D)$ $\{hypothesis\}$
$= \mathit{if}\ b\ ((C_{01} \parallel D); (C_2 \parallel D))\ \mathit{else}\ ((C_{02} \parallel D); (C_2 \parallel D))\{(L7)\}$
$= (\mathit{if}\ b\ (C_{01} \parallel D)\ \mathit{else}\ (C_{02} \parallel D)); (C_2 \parallel D)$ $\{(L15)\}$
$= RHS$

(5). $C_1 = b * C_0$, where $C_0 \in$ *CP(B)*.

We define *F(X)* $=_{df}$ $\mathit{if}\ b\ (C_0; X)\ \mathit{else}\ \mathit{skip}$, and $\{F^n(chaos), n \geq 0\}$ as $F^0(chaos) =_{df} chaos$, and $F^{n+1}(chaos) =_{df} F(F^n(chaos))$, for $n \geq 0$.

Then $C_1 = \mu X \bullet F(X) = \bigsqcup_{n\geq 0} F^n(chaos)$, and $F^n(chaos) \in$ *CP(B)*, for $n \geq 0$.

LHS
$= ((\bigsqcup_{n\geq 0} F^n(chaos)); C_2) \parallel D$ $\{continuity\ of\ \parallel, ;\}$
$= \bigsqcup_{n\geq 0}((F^n(chaos); C_2) \parallel D)$ $\{hypothesis\}$
$= \bigsqcup_{n\geq 0}((F^n(chaos) \parallel D); (C_2 \parallel D))$ $\{continuity\ of\ \parallel, ;\}$
$= RHS$ □

Corollary 5.2 If $C \in$ *CP(B)*, then $(b{*}C) \parallel D = b{*}(C \parallel D)$.

The proof is presented in [12]. □

6. Syntax-based Splitting Rules

This section discusses program splitting rules. First we show how the static analysis affects the partition of primitive commands into hardware and software components. Secondly we demonstrate how to construct hardware and software parts of a construct from those of its constituents. We establish the correctness of those rules by using the algebraic laws given in Section 3.

We introduce a predicate *Split*, which will be of great help in formalising the decomposition rules.

Definition 6.1 (*Split*)

Let $B = \{r_j, a_j \mid j \in I\}$. Given a sequential process *S*, its hardware/software partition (C, D) is specified by the following predicate:

$Split_B(S, C, D) =_{df}$
$S \sqsubseteq (C \parallel D) \wedge C \in CP(B) \wedge D \in H(B)$
$Var(C) \cap Var(D) = \emptyset \wedge Chan(C) \cap Chan(D) = B \wedge$
$InputChan(C) \cap Inputchan(D) = \emptyset \wedge$

Outputchan$(C) \cap$ *Outputchan*$(D) = \emptyset$
where *InputChan*(C) is the set of channels employed by C and only used for input tasks, *Outputchan*(C) is similar. □

6.1. The Bottom-up Splitting Approach

The *bottom-up* approach builds the hardware component from a program directly from the static analysis in one step, i.e., the hardware device is to provide all the services frequently used by the program. However, it constructs the software component from those of its constituents using the following rules.

Bottom-up Rule for Sequential Composition

$$\frac{\begin{array}{c}Split_B(S_i,\ C_i,\ D),\ i=1,2\\ Var(S_1)=Var(S_2),\ Chan(C_1)=Chan(C_2)\end{array}}{Split_B(S_1; S_2,\ C_1; C_2,\ D)}$$

Proof
$S_1\ ;\ S_2$ $\{;\ \textit{is monotonic}\}$
$\sqsubseteq\ (C_1 \parallel D);\ (C_2 \parallel D)$ $\{Th.5.1\}$
$=\ (C_1;\ C_2) \parallel D$ □

Bottom-up Rule for Conditional

$$\frac{\begin{array}{c}Split_B(S_i,\ C_i,\ D),\ i=1,2\\ Var(S_1)=Var(S_2),\ Chan(C_1)=Chan(C_2)\\ Var(b)\subseteq Var(C_1)\end{array}}{Split_B(\textit{if } b\, S_1\, else\, S_2,\ \textit{if } b\, C_1\, else\, C_2,\ D)}$$

Proof
if $b\, S_1\, else\, S_2$ $\{\textit{cond is monotonic}\}$
$\sqsubseteq$ *if* $b\ (C_1 \parallel D)\ else\ (C_2 \parallel D)$ $\{(L15)\}$
$=\ (\textit{if } b\, C_1\, else\, C_2) \parallel D$ □

Bottom-up Rule for Iteration

$$\frac{\begin{array}{c}Split_B(S,\ C,\ D)\\ Var(b)\subseteq Var(C)\end{array}}{Split_B(b*S,\ b*C,\ D)}$$

When $Var(b) \cap Var(D) \neq \emptyset$, we will introduce a local variable lb, and rewrite the conditional and iteration into the forms

$\mathbf{var}\ lb \bullet (lb := b;\ \textit{if } lb\, S_1\, else\, S_2)$, and

$\mathbf{var}\ lb \bullet (lb := b;\ lb * (S;\ lb := b))$ respectively by law DL1 and DL2. The partitioning rule for $lb := b$ will be discussed later.

The non-deterministic choice can be regarded as a special case of guarded choice when all the guards are *skip*. We present the partitioning rule for guarded choice constructs as follows and omit the rule for non-deterministic choice.

Bottom-up Rule for Guarded Choice

$$\frac{\begin{array}{c}Split_B(S_i,\ C_i,\ D),\ i=1,2\\ Var(S_1)=Var(S_2),\ Chan(C_1)=Chan(C_2)\\ Var(g_i)\subseteq Var(C_1),\ i=1,2\\ Chan(g_i)\subseteq Chan(C_1),\ i=1,2\end{array}}{Split_B((g_1\ S_1)[\!](g_2\ S_2),\ (g_1\ C_1)[\!](g_2\ C_2),\ D)}$$

The proofs for the last two rules are straightforward and are presented in [12], due to the page limit.

6.2. The Top-down Splitting Approach

In this approach, both the hardware and software components of the source program are assembled from those of its constituents.

Before presenting a set of *top-down* splitting rules, we introduce the notion of *interface-consistency* on hardware components.

Definition 6.2 For $k = 1, 2$, let
$D_k =_{df} \mu X \bullet ([\!]_{i\in I_k}(r_i?x_i; M_i; a_i!y_i; X)[\!]skip)$,
D_1 and D_2 are said to be interface-consistent, denoted by *Consist*(D_1, D_2), if *Var*(D_1) = *Var*(D_2), and
Chan$(D_1)\backslash B_1$ = *Chan*$(D_2)\backslash B_2$,
where $B_k =_{df} \{r_j, a_j \mid j \in I_k\}$, for $k = 1, 2$.

In such a case, we define
$D = union(D_1, D_2) =_{df}$
$\mu X \bullet ([\!]_{i\in I_1\cup I_2}(r_i?x_i; M_i; a_i!y_i; X)[\!]skip)$ □

We first present a basic rule, from which and the *bottom-up* rules we obtain the corresponding *top-down* rule straightforwardly in each case.

Rule for Hardware Augmentation

$$\frac{\begin{array}{c}Split_{B_1}(S,\ C,\ D_1)\\ Consist(D_1, D_2),\ Chan(C)\cap B_2\subseteq B_1\end{array}}{Split_{B_1\cup B_2}(S,\ C,\ D)}$$

Top-down Rule for Sequential Composition

$$\frac{\begin{array}{c}Split_{B_i}(S_i,\ C_i,\ D_i),\ i=1,2\\ Var(S_1)=Var(S_2),\ Chan(S_1)=Chan(S_2)\\ Consist(D_1, D_2)\end{array}}{Split_{B_1\cup B_2}(S_1; S_2,\ C_1; C_2,\ D)}$$

Top-down Rule for Conditional

$$\frac{\begin{array}{c}Split_{B_i}(S_i,\ C_i,\ D_i),\ i=1,2\\ Var(S_1)=Var(S_2),\ Chan(S_1)=Chan(S_2)\\ Consist(D_1, D_2),\ Var(b)\subseteq Var(C_1)\end{array}}{Split_{B_1\cup B_2}(\textit{if } b\, S_1\, else\, S_2,\ \textit{if } b\, C_1\, else\, C_2,\ D)}$$

Top-down Rule for Guarded Choice

$$\frac{\begin{array}{c}Split_{B_i}(S_i,\ C_i,\ D_i),\ i=1,2\\ Var(S_1)=Var(S_2),\ Chan(S_1)=Chan(S_2)\\ Consist(D_1, D_2)\\ Var(g_i)\subseteq Var(C_1),\ Chan(g_i)\subseteq Chan(C_1),\ i=1,2\end{array}}{Split_{B_1\cup B_2}((g_1\ S_1)[\!](g_2\ S_2),\ (g_1\ C_1)[\!](g_2\ C_2),\ D)}$$

6.3. Splitting Primitive Commands

This section deals with primitive commands splitting. We only investigate the following nontrivial cases: the assignment, the invocation of a procedure, and the annotated blocks.

1. An assignment $u := e(v)$

We focus on the cases where both hardware and software participate in the evaluation of $e(v)$ and the update of u.

Case 1: $e(v)$ is a "busy" expression, and the variable v has been allocated to the hardware component.

$Split_B(u := e(v),\ C,\ D)$, where
$C =_{df} (r_j\,!\,1;\ a_j\,?\,u)$, and
$D =_{df} \mu X \bullet ((r_j\,?\,x;\ y := e(v);\ a_j\,!\,y; X) \parallel skip)$

Case 2: $e(v)$ is a "busy" expression, however, v has been allocated to the software component.

$Split_B(u := e(v),\ C,\ D)$, where
$C =_{df} (r_j\,!\,v;\ a_j\,?\,u)$, and
$D =_{df} \mu X \bullet ((r_j\,?\,x;\ y := e(x);\ a_j\,!\,y; X) \parallel skip)$

Case 3: $e(v)$ is not a "busy" expression, but v is allocated to the hardware component.

$Split_B(u := e(v),\ C,\ D)$, where
$C =_{df} (\mathbf{var}\ lv \bullet (r_j\,!\,1;\ a_j\,?\,lv;\ u := e(lv)))$, and
$D =_{df} \mu X \bullet ((r_j\,?\,x;\ y := v;\ a_j\,!\,y; X) \parallel skip)$ □

More intricate case of assignment $u := e(v, w)$, where v and w have respectively been allocated to the software component and the hardware one, will be converted to several successive assignments owning the form we have dealt with above, by the algebraic law with respect to assignments.

2. A procedure invocation

Without lose of generality, we investigate the invocation $proc(e_S, e_H, v_S, v_H)$, where e_S is supplied by software, e_H is evaluated by hardware, v_S and v_H are allocated to software and hardware, respectively. We are interested in the case where the procedure is implemented by hardware.

$Split_B(proc(e_S, e_H, v_S, v_H),\ C,\ D)$, where
$C =_{df} (cr_j\,!\,e_S; ca_j\,?\,v_S)$, and
$D =_{df} \mu X \bullet ((cr_j\,?\,x; proc(x, e_H, y, v_H); ca_j\,!\,y; X) \parallel skip)$

3. An annotated block

We concentrate on the case where the block $\langle B(v_S, v_H)\rangle$ is predetermined to be implemented by hardware, and the variables that occur in the block v_S and v_H are allocated to software and hardware, respectively. We need to arrange the data flow between software and hardware.

$Split_B(\langle B(v_S, v_H)\rangle,\ C,\ D)$, where
$C =_{df} (cr_j\,!\,v_S; ca_j\,?\,v_S)$, and
$D =_{df} \mu X \bullet ((cr_j\,?\,x; y := x; B(y, v_H); ca_j\,!\,y; X) \parallel skip)$

7. Conclusion

This paper shows how the hardware/software partitioning problem can be tackled in the algebra of programs. The partitioning task consists of the static program analysis phase and the splitting phase, where the former provides the information for moving operations from software to hardware and reducing the communication between components, and the latter supports a compositional approach to the program partitioning. To synchronize software and hardware components, and reduce the complexity of their interface, we introduce a simple handshaking protocol, and propose a normal form for the hardware components. The correctness of the splitting process is verified using the algebraic laws of the source language. To deal with co-design of embedded systems, we shall introduce timing constraints into our source program, which will result in timed hardware and software components.

References

[1] A. Balboni *et al*, "Partitioning and Exploration Strategies in the TOSCA Design Flow", In *Proceedings of Fourth International Workshop on Hardware/Software Codesign*, 62–69, IEEE Computer Society Press, (1996).

[2] T. Cheung, "A Multi-level Transformation Approach to Hardware/Software Co-design", In *Proceedings of Fourth International Workshop on Hardware/Software Codesign*, 10–17, (1996).

[3] He Jifeng, *Provably Correct Systems: Modelling of Communication Languages and Design of Optimised Compilers*, McGraw-Hill Publisher, 1994.

[4] He Jifeng, I. Page and J. Bowen, "A Provable Hardware Implementation of Occam", *Lecture Notes in Computer Science* 711, 693–703, (1993).

[5] He Jifeng and J. Bowen, "Specification, Verification and Prototyping of an Optimised Compiler", *Formal Aspect of Computing* 6, 643–658, (1994).

[6] He Jifeng *et al*, "Provably Correct Systems", *Lecture Notes in Computer Science* 863, 288–335, (1994).

[7] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.

[8] C.A.R. Hoare and He Jifeng, *Unifying Theories of Programming*, Prentice Hall, 1998.

[9] C.A.R. Hoare *et al*, "Laws of Programming", *Communications of the ACM*, Vol 30(8): 672-686, 1987.

[10] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin, *Principles of Program Analysis*, Springer-Verlag, 1999.

[11] Ian Page and Wayne Luk, "Compiling Occam into FPGAs", in *FPGAs*, eds., Will Moore and Wayne Luk, 271-283, Abingdon EE&CS books, 1991.

[12] Qin Shengchao and He Jifeng, "An Algebraic Approach to Hardware/software Partitioning", UNU/IIST Report 206, Macau, June, 2000.

[13] A.W.Roscoe and C.A.R. Hoare, "Laws of Occam Programming", *Theoretical Computer Science*, Vol 60: 177-229, 1988.

[14] Augusto Sampaio, "An Algebraic Approach to Compiler Design", *World Scientific*, (1997).

[15] L. Silva, A. Sampaio and E. Barros, "A Normal Form Reduction Strategy for Hardware/software Partitioning", *Formal Methods Europe (FME) 97, Lecture Notes in Computer Science, 1313*, (1997) 624-643.

8. Appendix

The source code in Example 4.3.

– ATM switch
– Variables
– GFC, VPI, VCI, PT, CLP, HEC, Pl - ATM fields
– aT - arrival time of the cell
– nVPI, nVCI - new id of the cell
– QoS - quality of service
– pN - Port number
– HPP - variables of high priority peak policy
– LPP - variables of low priority peak policy
– HPM - variables of high priority mean policy
– LPM - variables of low priority mean policy
– RP/nRP - current/new value of rejected cells due to peak
– RM/nRM - current/new value of rejected cells due to mean
– RC/nRC - current/new value of rejected cells
– AC/nAC - current/new value of accepted cells
– send - boolean variable which decides whether the cell must be sent or not.
– Communication with the environment (channels)
– chCell - receives the cell
– chOut - sends the cell
– ch1ReadTable - request data from the table
– ch2ReadTable - receive the fields of the table
– chRouteTable - receive the new identifiers and output port
– chWTable - update the table

– Generic Cell Rate Algorithm — Leaky Bucket
– X = bucket level
– LCT = Last Conformance Time
– ta = arrival time
– I = increment
– L = cell delay variation tolerance

---

procedure GCRA(**in** X, LCT, at, I, L: **int**,
out ok: **Bool**, nLCT, nX: **int**)
begin
var Xtmp: **int** • Xtmp := X - (at - LCT);
if (Xtmp < 0) nX := I; nLCT := at; ok := *true*;
else if (Xtmp ≤ L) nX := Xtmp + I; nLCT := at; ok := *true*;
else nX := X; nLCT := LCT; ok := *false*;
end
procedure UPT(**in** tt: **Bool**, p,m,r,a,cl: **int**,
out send: **Bool**, nRP, nRM, nRC, nAC, nCLP: **int**)
begin
send := tt; nRP := p; nRM := m;
nRC := r; nAC := a; nCLP := cl;
end

---

var GFC, VPI, VCI, PT, HEC, Pl, aT, QoS: **int**, X, L,
I, LCT, nX, nLCT: **record of** HPP, LPP, HPM, LPM: **int end**,
ok, CLP, nCLP: **record of** HPP, LPP, HPM, LPM: **Bool end**,
RP, nRP, RM, nRM, RC, nRC, AC, nAC: **int**, send: **Bool**,
nVPI[3], nVCI[3], pN[3]: **array of int**;
– Read the cell and the table
chCell ? (GFC, VPI, VCI, PT, CLP, HEC, Pl, aT);
ch1ReadTable ! (VPI, VCI);
ch2ReadTable ? (QoS, X, L, I, LCT, RP, RM, RC, AC);
chRouteTable ? (nVPI[0], nVCI[0], pN[0]);
chRouteTable ? (nVPI[1], nVCI[1], pN[1]);
chRouteTable ? (nVPI[2], nVCI[2], pN[2]);
GCRA(X.HPP, LCT.HPP, aT, I.HPP, L.HPP,
ok.HPP, nLCT.HPP, nX.HPP);
GCRA(X.HPM, LCT.HPM, aT, I.HPM, L.HPM,
ok.HPM, nLCT.HPM, nX.HPM);
GCRA(X.LPP, LCT.LPP, aT, I.LPP, L.LPP,
ok.LPP, nLCT.LPP, nX.LPP);
GCRA(X.LPM, LCT.LPM, aT, I.LPM, L.LPM,
ok.LPM, nLCT.LPM, nX.LPM);
if CLP
if ¬ ok.LPP UPT(*false*,RP+1,RM,RC+1,AC,CLP,send,
nRP,nRM,nRC,nAC,nCLP);
else if ok.LPM UPT(*true*,RP,RM,RC,AC+1,CLP,
send,nRP,nRM,nRC,nAC,nCLP);
else UPT(*false*,RP,RM+1,RC+1,AC,CLP,
send, nRP,nRM,nRC,nAC,nCLP);
else if ok.HPP ∧ ok.HPM UPT(*true*,RP,RM,RC,AC+1,CLP,
send, nRP,nRM,nRC,nAC,nCLP);
else if ok.HPP ∧ ¬ ok.HPM ∧ ok.LPM
UPT(*true*,RP,RM,RC,AC+1,1,send,
nRP,nRM,nRC,nAC,nCLP);
else if ok.HPP ∧ ¬ ok.HPM ∧ ¬ ok.LPM
UPT(*false*,RP,RM+1,RC+1,AC,1,send,
nRP,nRM,nRC,nAC,nCLP);
else if ¬ ok.HPP ∧ ok.LPP ∧ ok.LPM
UPT(*true*,RP,RM,RC,AC+1,1,send,
nRP,nRM,nRC,nAC,nCLP);
else if ¬ ok.HPP ∧ ¬ ok.LPP
UPT(*false*,RP+1,RM,RC+1,AC,1,send,
nRP,nRM,nRC,nAC,nCLP);
else if ¬ ok.HPP ∧ ¬ ok.LPP ∧ ¬ ok.LPM
UPT(*false*,RP,RM+1,RC+1,AC,1,send,
nRP,nRM,nRC,nAC,nCLP);
else skip;
– Send the cell
if send
chOut ! (pN[0],QoS,GFC,nVPI[0],nVCI[0],PT,nCLP, HEC,Pl);
chOut ! (pN[1],QoS,GFC,nVPI[1],nVCI[1],PT,nCLP, HEC,Pl);
chOut ! (pN[2],QoS,GFC,nVPI[2],nVCI[2],PT,nCLP, HEC,Pl);
else skip;
– Update the table
chWTable ! (nX.HPP,nLCT.HPP,nX.HPM,
nLCT.HPM,nX.LPP, nLCT.LPP); □